Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica

Un compilatore per un linguaggio per smart contract intrinsecamente tipato

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Stefano Bucciarelli

II Sessione
Anno Accademico 2018-2019

Indice

Capitolo 1

Introduzione

Da sempre, un elemento portante della società è lo scambio di risorse, che ha condizionato ogni epoca. Normalmente concludere uno scambio prevede la figura di un garante che impone delle regole affinché lo scambio sia valido. Per la prima volta nella storia si hanno a disposizione i mezzi per eliminare questa figura, perché si ha un sistema in cui l'esistenza stessa di un contratto valida la transazione. L'informatica attraverso la programmazione permette di realizzare questo tipo di contratti, che nell'ambito vengono chiamati *smart contract*.

Gli *smart contract* devono garantire le proprietà di fiducia, affidabilità e di sicurezza, che in precedenza erano delegate al garante, ma che adesso diventano possibili grazie alle blockchain. Queste ultime sono code nelle quali è consentita l'operazione di lettura, mentre l'unica operazione di scrittura è l'aggiunta (o transazione): non è possibile quindi modificare un elemento già presente nella blockchain; di conseguenza viene mantenuto un registro con la storia di tutte le transazioni eseguite. Inoltre questa struttura dati è distribuita, quindi è parte di una rete peer-to-peer in cui viene replicata per ogni nodo. Di conseguenza le istanze di *smart contract*, se memorizzati nella blockchain, diventano programmi che possono essere eseguiti in modo distribuito e sicuro, potendo controllare lo scambio di denaro tra diverse parti.

A livello di programmazione uno *smart contract* è definito da un identificativo univoco, da uno stato – cioè i campi – e dal suo codice, inteso come insieme di metodi che modificano lo stato del contratto. Per implementare *smart contract* vi sono a disposizioni linguaggi di programmazione dedicati, a ognuno dei quali è associata la relativa blockchain: ad esempio troviamo il linguaggio Solidity per la blockchain Ethereum o Liquidity per Tezos.

Dal momento che è possibile gestire considerevoli somme di denaro, il codice dei contratti è codice critico: diventa quindi interessante fare analisi statica sul comportamento degli *smart contract*. In merito sono stati fatti diversi lavori; di questi ne è particolarmente

interessante uno (Laneve, Coen, and Veschetti 2019) in cui emerge che, nel momento in cui un utente razionale agisce per minimizzare le perdite di denaro (o equivalentemente per massimizzare il guadagno) lo *smart contract* può forzarlo a prendere determinate decisioni. Per esempio, chiedendo all'utente una cauzione che il contratto restituirà solo in caso di suo comportamento onesto, questi se vorrà massimizzare il guadagno si vedrà costretto a rispettare il contratto. È inoltre possibile verificare algoritmicamente il *comportamento emergente* del sistema: ogni utente che è egoisticamente interessato a massimizzare il suo guadagno non impedisce agli altri utenti di fare lo stesso.

Questa analisi è stata fatta definendo un linguaggio chiamato `scl` (Smart Calculus Language), che, con un ristretto insieme di funzioni sugli *smart contract*, permette di modellare il comportamento di contratti e utenti (umani). `Scl` è un linguaggio ad attori object-based, che rende più semplice l'analisi di *smart contract* – ma allo stesso tempo espressivo abbastanza da essere Turing completo: permette l'invocazione di metodi, la modifica dei campi, il comportamento condizionale, la ricorsione, il sollevamento di eccezioni. Può essere scritto un programma in `scl` che corrisponde a un modello dove gli utenti e i contratti interagiscono tra loro. L'analisi associa un'unica formula nell'aritmetica di Presburger che descrive lo spazio delle possibili scelte, elaborandola si ottiene la strategia dell'utente per massimizzare il profitto.

`Scl` precedentemente era un linguaggio solamente teorico: un programma in questo linguaggio non poteva essere eseguito per la mancanza degli strumenti necessari. Il mio personale contributo, dunque, è consistito nel produrre questi strumenti per rendere effettivamente possibile l'uso di `scl` in ulteriori analisi. Ho quindi definito un parser per dare una sintassi a `scl`, e successivamente ho implementato un compilatore verso un linguaggio per *smart contract* già conosciuto. L'obiettivo ultimo è stato quello di permettere la scrittura e il deploy di *smart contract* su una blockchain esistente usando solo `scl`.

1.1 Presentazione del lavoro svolto

In questa tesi mostriamo le tecniche usate, le caratteristiche implementative e la risoluzione ai problemi riscontrati che hanno portato alla realizzazione di questi tool di supporto al linguaggio `scl`.

In particolare nel capitolo 2 introduciamo la questione del *type checking*, che un compilatore generico deve affrontare per controllare che i vincoli di tipo nel programma da compilare siano rispettati. In merito, definiamo una *Intrinsically Type Data Structure*, che permette la realizzazione di una strategia per fare *type checking*. Questa tecnica prevede l'uso di un albero di sintassi astratta in cui la sua stessa struttura cattura sintatticamente tutti e soli i programmi ben tipati, permettendo al compilatore di non

effettuare un controllo sui tipi dopo la costruzione dell'albero. L'implementazione di una *Intrinsically Typed Data Structure* è possibile in pochi linguaggi funzionali: tra questi ci concentreremo su OCaml – che è anche il linguaggio che ho usato per la realizzazione di parser e compilatore – vedendo quali strumenti mette a disposizione.

Nel capitolo 3 vediamo le caratteristiche principali del linguaggio `scl`: notiamo in particolare che il linguaggio non presentava una sintassi che permettesse di scrivere in maniera semplice i programmi. Viene mostrata quindi la grammatica che ho introdotto, necessaria al parser per costruire l'albero di derivazione nel linguaggio. Vediamo anche che l'albero di sintassi astratta era già definito da `scl` come una struttura intrinsecamente tipata, quindi il parser non si deve preoccupare di eseguire un controllo di tipi sull'albero, avendo comunque la garanzia di rispettare i vincoli di tipo.

Una volta definite le strutture in input e in output del parser, nel capitolo 4 mostriamo l'aspetto implementativo. Mi sono in primo luogo preoccupato dell'analisi lessicale, in modo che i singoli caratteri in input fossero raggruppati in token. Poi ho implementato l'analizzatore sintattico che sui token in ingresso costruisce l'albero di sintassi astratta. Il parser è stato implementato grazie alla tecnica dei *parser combinator*, strategia che consiste nel definire funzioni di ordine superiore che prendendo dei parser in input, restituendo un nuovo parser che è il risultato della composizione degli input.

Dopo aver mostrato il parser vediamo nel capitolo 5 il compilatore, il cui scopo è di rendere possibile l'implementazione di un programma `scl` su una blockchain. Detto compilatore traduce nel linguaggio per *smart contract* Solidity del linguaggio `scl`. Essendo `scl` un linguaggio molto più povero di costrutti rispetto a Solidity, la sua compilazione non produce tutti i possibili costrutti di quest'ultimo linguaggio, ma solo un loro sottoinsieme. In più, vedremo che vengono tradotti solo i contratti definiti in `scl`, poiché Solidity non dispone dei mezzi necessari per la definizione degli utenti. Il compilatore in primo luogo traduce l'albero di sintassi astratta di `scl` nell'analogo albero in Solidity; successivamente da quest'ultimo genera l'effettivo codice Solidity. L'albero di sintassi astratta di Solidity generato dalla compilazione è intrinsecamente tipato: il compilatore, quindi deve far associare ad ogni programma ben tipato in `scl` lo stesso programma ben tipato in Solidity. Questo avviene creando una corrispondenza di tipi di dato in `scl` e i tipi di Solidity – una volta fatto questo diventa poi possibile tradurre tutti i costrutti del linguaggio `scl` nei loro costrutti corrispondenti.

La sola compilazione in codice Solidity non è sufficiente a permettere la creazione sulla blockchain dei contratti. Nel capitolo 6 mostriamo come il compilatore generi anche uno script Python che, collegandosi alla rete, istanzi i contratti permettendo quindi di fare il deploy di questi ultimi sulla blockchain.

Capitolo 2

Intrinsically Typed Data Structure

2.1 Introduzione

Durante la compilazione verso un linguaggio qualsiasi, il compilatore si deve preoccupare del *type checking* cioè di quell'operazione che verifica che i vincoli di tipo nel programma vengano rispettati. Ad esempio un vincolo di tipo si ha in un assegnamento, dove è richiesto che il tipo del *left-hand side* (*lhs*) sia compatibile con il tipo del *right-hand side* (*rhs*), altrimenti si può incorrere in diversi errori durante l'esecuzione.

Per garantire la coerenza di tipi nel parser e nel compilatore, invece che usare le classiche tecniche, è stata scelta una tecnica emergente presente in solo pochi linguaggi funzionali (tra cui OCaml), che consiste nell'uso di una *Intrinsically Typed Data Structure*.

2.2 Approccio al type checking

Tradizionalmente il *type checking* avviene nella fase di analisi semantica del processo di compilazione – quindi dopo che è avvenuto il parsing – una volta che è stato costruito l'albero di sintassi astratta (AST). Si ha così che il *type checking* avviene esternamente all'albero di derivazione: questo è infatti sottoposto a un controllo, dove per ogni sottoalbero viene valutato il tipo e in ogni nodo viene verificato che i tipi dei relativi sottoalberi combacino.

Nel nostro caso però, sia nel compilatore che nel parser, abbiamo sfruttato una tecnica differente in cui l'albero di sintassi astratta è un *Intrinsically Typed Data Structure*: questo significa che l'albero è una struttura dati con tipo dipendente – il tipo è in relazione ai termini all'interno della struttura – che permette di esprimere tutti e i soli programmi ben tipati. È quindi la struttura stessa che definisce che cosa è ben tipato: non abbiamo

bisogno di implementare un *type checker* separato che analizzi l'albero poiché i vincoli di tipo sono già all'interno della definizione stessa dell'albero. Il *type checking*, quindi, avviene direttamente nel momento della costruzione dell'albero, facendo comunque rispettare i vincoli di tipo, ma in modo più immediato e con un'implementazione più semplice.

Per implementare parser e compilatore, siamo interessati a implementare un *Intrinsically Typed Data Structure* in OCaml, vediamo quindi gli strumenti che il linguaggio mette a disposizione.

2.3 Generic Algebraic Data Type

In prima battuta capiamo cosa sia un *Algebraic Data Type* (ADT) che OCaml permette di definire. Un ADT è un tipo composto definito tramite intersezione o unione disgiunta di altri tipi. Mostriamo un esempio di ADT:

```
type expr =
  | Int of int
  | Bool of bool
  | And of expr * expr
  | Plus of expr * expr
  | Eq of expr * expr
```

In questo esempio un tipo `expr` può essere `Int` di un intero, `Bool` di un booleano oppure `Plus`, `And` o `Eq` di due altre `expr`. Si ha quindi un'unione disgiunta di vari costruttori (che sono `Int`, `Bool`, `Plus`…), e all'interno di alcuni si ha un'intersezione – ad esempio all'interno di `Plus` si trova una coppia di `expr`. Notiamo però, che, sempre nell'esempio, se si costruisse un'espressione `And((Int 9),(Bool true))` questa sarebbe un'espressione sintatticamente corretta, ma non lo sarebbe semanticamente: non sarebbe infatti ben tipata, siccome l'`And` si può fare solo tra tipi booleani. È necessario, quindi che quando viene valuta l'espressione ci sia un controllo di tipo. La funzione che si preoccupa della valutazione sarebbe:

```
let rec eval : expr -> int_or_bool =
function
  | Int n -> I n
  | Bool b -> B b
  | And(e1,e2) ->
  (match eval e1, eval e2 with
     B n, B m -> B (n && m)
     | _, _ -> raise TypeError)
  | ...
```

In questo modo scrivendo `And((Int 9),(Bool true))` non si riscontrerebbe nessun errore a tempo di compilazione, ma se si cercasse di calcolare `eval (And((Int 9),(Bool true)))` allora sarebbe sollevata un'eccezione durante l'esecuzione.

Sempre con l'obiettivo di avere delle strutture *Intrinsically typed* siamo interessati a un costrutto in cui la sintassi della struttura stessa possa permettere di scrivere espressioni solo ben tipate (il concetto stesso di ben tipato è in realtà definito dalla stessa struttura). OCaml infatti mette a disposizione anche *Generalized Algebraic Data Type* (GADT), che come suggerisce il nome è la generalizzazione degli ADT. Come d'altronde era già permesso negli ADT, i GADT permettono di definire dei tipi parametrici – ma in più si ha la possibilità di vincolare sintatticamente il parametro al momento della sua istanziazione con il tipo di ritorno dei costruttori del GADT (che dovrà essere specificato).

```
type 'a expr =
| Int : int -> int expr
| Bool : bool -> bool expr
| And : bool expr * bool expr -> bool expr
| Plus : int expr * int expr -> int expr
| Eq : 'a expr * 'a expr -> bool expr
```

In questo esempio, quindi, l'espressione `And((Int 9),(Bool true))` risulterebbe mal tipata in quanto non conforme alla struttura appena definita: il costruttore `And` richiederebbe una coppia di espressioni booleane, ma il primo elemento della coppia `Int 9` è definito come un'espressione intera. Scrivendo infatti `And((Int 9),(Bool true))`, sarebbe sollevato un errore a tempo di compilazione. Definendo così il tipo `expr` anche la valutazione del tipo risulta più semplice rispetto all'analoga nel caso del ADT, infatti non ci si deve preoccupare di nessun controllo di tipo:

```
let rec eval: : type a. a expr -> a =
function
  | Int n -> n
  | Bool b -> b
  | And(e1,e2) -> (eval e1) && (eval e2)
  | ...
```

Abbiamo quindi che la struttura del tipo `expr` definita con l'uso degli GADT vincola sintatticamente a scrivere espressioni ben tipate: ciò significa che `expr` è intrinsecamente tipata. Quindi l'implementazione dell'albero di sintassi astratta, sarà del tutto analoga a quella di `expr`, dove i costruttori, invece di essere relativi alle espressioni saranno relativi ai costrutti di un programma nel linguaggio.

2.4 Conclusione

Abbiamo visto come i GADT permettono la definizione di una *Intrinsically Typed Data Structure*: usando questa tecnica nell'implementazione del parser e del compilatore realizzato, abbiamo evitato di eseguire un *type checking* ridondante e con buone probabilità che presentasse bug, avendo la certezza di produrre codice ben tipato.

Capitolo 3

Smart Calculus Language

3.1 Introduzione

In questo capitolo iniziamo a descrivere il parser, senza analizzare, per ora, l'aspetto implementativo. Descriviamo in un primo luogo `scl`, il linguaggio verso il quale viene fatto il parsing, successivamente definiamo la grammatica che descrive una sintassi per `scl`, e infine per tale grammatica mostriamo il relativo albero di sintassi astratta tipato generato dal parser, in maniera da definire chiaramente la struttura dell'input e dell'output del parser.

3.2 Descrizione del linguaggio scl

`Scl` è un linguaggio imperativo ad attori (come per esempio `Erlang`) che permette di descrivere il comportamento di contratti e umani – cioè gli utenti che interagiscono sui contratti. `Scl` è un linguaggio usato per l'analisi di *smart contract*, quindi, con l'obiettivo di fare un'analisi più mirata, semplice ed essenziale, è stato reso volutamente minimale. Nonostante ciò `scl` è un linguaggio Turing completo, in quanto permette l'assegnamento, l'istruzione condizionale, l'invocazione di funzioni, la ricorsione e l'esecuzione di transazioni (con significato nel contesto delle basi di dati).

Un programma `scl` consiste in una una configurazione, ovvero un insieme di attori (contratti o umani) che vengono definiti con campi e metodi, in maniera analoga ai linguaggi di programmazione a oggetti. Si ha che ogni attore conosce tutti gli altri attori della configurazione, rendendo possibile – per esempio – che un umano chiami un metodo di uno specifico contratto, senza avere bisogno di parametri o campi aggiuntivi.

Sebbene siano due entità distinte, il codice dei contratti e degli umani è molto simile, la differenza pregnante è che un contratto può sollevare un'eccezione, mentre l'umano può osservare il fallimento dei contratti catturando l'eccezione e prendendo decisioni di conseguenza. Gli umani hanno anche a disposizione l'operazione di scelta, che consiste in un operatore non deterministico con il quale non si conosce a priori quale codice l'umano andrà ad eseguire. L'idea è che nella realtà l'umano non ha un comportamento deterministico e perciò si devono prendere in considerazioni tutte le sue azioni possibili; proprio questa diventa la parte interessante dell'analisi con `scl` (Laneve, Coen, and Veschetti 2019), in quanto si cerca di capire quale comportamento sarà più vantaggioso per l'umano. Inoltre per gli umani è necessario definire i comandi iniziali, che descrivono il loro comportamento e le loro interazioni con gli altri attori della configurazione.

Anche se non era prevista una vera e propria sintassi, all'inizio di questa tesi era già prevista una struttura per l'albero di sintassi astratta tipato in `scl` (era infatti implementato in OCaml con i GADT). È stato, quindi, implementato il parser avendo come albero di sintassi astratta in output direttamente queste strutture, senza passare per una struttura intermedia. Minor sforzo ha richiesto la fase di implementazione, prestando infatti il vantaggio di non preoccuparsi di incorrere in errori di tipo durante il parsing, dal momento che la stessa esistenza della struttura in output garantisce che il programma in input sia ben tipato.

3.3 Sintassi per scl

Il parser verso `scl` deve costruire l'albero di sintassi astratta del testo in input, basandosi sulla sua grammatica specifica, la quale necessita di essere definita.

Nella tabella 3.1 mostriamo la grammatica su cui si basa l'analizzatore sintattico, nella quale sono presenti i simboli terminali che necessitano di essere riconosciuti durante l'analisi lessicale: che sono `Int` (ovvero l'insieme dei numeri interi), `String` (l'insieme delle stringhe) e `Var` (l'insieme di tutte le variabili), a cui si aggiungono tutte le parole chiave (ad esempio **return**, **if**, $+$, $>$, …). Invece la parte di analisi sintattica, una volta riconosciuti i simboli terminali, si preoccuperà di riconoscere i simboli non terminali e le opportune produzioni ricalcando la struttura del programma in input.

Questa sintassi permette di definire un insieme di attori, ognuno dei quali ha sia dei campi, che devono essere dichiarati, che una lista di metodi. Questi ultimi sono composti da una *signature* (con tipo in input e in output, dove quest'ultimo è sempre presente), uno statement (consistente in una lista di comandi di assegnamento, o istruzione condizionale o, nel caso di attori umani, scelta) ed espressione di ritorno. Nella grammatica vengono specificati i tipi di `scl`: interi, booleani, stringhe, contratti e umani. Sono messe a disposizione le principali operazioni tra questi, in particolar modo per gli interi e per

conf	::=	*act**	(configurazione)
act	::=	(*Human* | *Contract*) **(** Int **)**? **{** *decl** *meth** (*stm* **return** *e*)?**}**	(attore)
stm	::=	*decl* | Var **=** *rhs* | **if** *be* **then** *stm* **else** *stm* | *stm stm* | *stm* **+** *stm* | **{** *stm* **}**)	(statement)
decl	::=	*t* Var (**=** *v*)?	(dichiarazione)
meth	::=	**function** Var **(**(*t* Var (**,** *t* Var)*)? **):** *t* **{** *stm* **return** e **}**	(metodo)
rhs	::=	*e* | (ce **.**)? Var (**. value (** *ie* **)**)? **(** (e (**,** e)*)? **)**	(right-hand side)
v	::=	Int | *Bool* | String	(valore)
Bool	::=	**true** | **false**	(booleano)
ie	::=	Int | Var | **fail** | *ie*? **-** *ie* | *ie* **+** *ie* | Int ***** *ie* | **max (** *ie* **,** *ie* **)** | **symbol (** String **)** | **(** iexp **)**	(espressione intera)
be	::=	*Bool* | Var | **fail** | *ie* **>** *ie* | *ie* **>=** *ie* | *ie* **<** *ie* | *ie* **<=** *ie* | *e* **==** *e* | **!** *be* | *be* **&&** *be* | *be* **||** *be* | **(** *be* **)**	(espressione booleana)
se	::=	String | Var | **fail**	(espressione stringa)
ce	::=	this | Var | **fail** | **contr_addr (** String **)**	(espressione contratto)
he	::=	Var | **fail** | **hum_addr (** String **)**	(espressione umano)
e	::=	*ie* | *be* | *se* | *ce* | *he*	(espressione)
t	::=	**int** | **bool** | **string** | **Contract** | **Human**	(tipo)

Tabella 3.1: BNF per il parser, dove *conf* è il simbolo iniziale

i booleani. È possibile anche assegnare una stringa a un intero tramite il costrutto **symbol**, che similmente alle macro di C indica un valore costante.

Particolarmente interessante è il fatto che nella dichiarazione dell'attore si può specificare tra () il *balance* iniziale, ovvero la quantità di criptovaluta che è inizialmente assegnata al contratto o all'umano. Ciò equivale a fare un assegnamento alla variabile intera denominata `balance`. È inoltre possibile inviare dei quantitativi di criptovaluta al contratto nelle chiamate di funzioni semplicemente con la parola chiave **value** e specificando il valore. Come mostriamo nel sottostante breve esempio: cercando di derivare `x = y` partendo dal non terminale *stm* possiamo facilmente notare che la grammatica scelta è ambigua:

```
stm -> Var = rhs -> x = rhs -> x = e
```

uno statement, quindi diventa un assegnamento di un'espressione a una variabile. L'espressione ora può avere due produzioni:

Caso 1: l'espressione generica diventa un'espressione intera

```
x = ie -> x = y
```

Caso 2: l'espressione generica diventa un'espressione booleana

```
x = be -> x = y
```

Notiamo quindi che in entrambi i casi è stata prodotta la stessa sequenza partendo dallo stesso non terminale, ma con derivazioni diverse. L'ambiguità è quindi dovuta al fatto che diversi non terminali (tipicamente relativi alle espressioni) possono accettare indistintamente delle variabili a prescindere dal loro tipo. È necessario per cui che il parser quando incorre nel riconoscimento di una variabile riconosca direttamente il suo tipo. Affronteremo in seguito il problema.

3.4 Eliminazione ricorsione sinistra

Notiamo che la grammatica della tabella 3.1, oltre a essere ambigua è anche ricorsiva sinistra; come sarà più chiaro successivamente, il parser opererà in maniera ricorsiva, dobbiamo allora eliminare dalla grammatica questo tipo di ricorsione. Mostriamo soltanto il caso dell'eliminazione della ricorsione sinistra nel non terminale *ie* (connotazione per le espressioni intere), per gli altri non terminali infatti la risoluzione è analoga. L'idea è quella di sostituire al non terminale due non terminali: il primo con tutte e sole regole senza ricorsione sinistra il secondo con le regole in cui essa è presente.

Nella tabella 3.2 i due non terminali vengono rispettivamente denominati *atomic_ie* e *cont_ie*. Il primo, *atomic_ie*, rappresenta un non terminale per alcune espressioni

atomic_ie	::=	Int | Var | **fail** | **-** *ie* | **max** (*ie* , *ie*) | **symbol** String | (*ie*)
cont_ie	::=	+ *ie* | **-** *ie* | ***** *ie*
ie	::=	*atomic_ie cont_ie*?

Tabella 3.2: Regole di produzione per il non terminale *ie* senza ricorsione sinistra

intere (appunto quelle atomiche), mentre il secondo, *cont_ie*, considerato singolarmente non ha alcuna valenza: deve essere concatenato a un'espressione intera, che sarà quindi *atomic_ie*. Il non terminale *ie* viene costruito concatenando i due non terminali appena prodotti.

3.5 Albero di sintassi astratta

Dopo aver definito la grammatica, mostriamo l'albero di sintassi astratta generato come output dal parser, che rappresenterà la struttura logica del programma in input. Nel listato 3.1 mostriamo il tipo di ritorno per la grammatica della tabella 3.1. Notiamo come la grammatica definita precedentemente rimane molto conforme all'albero di sintassi astratta. Infatti ai simboli non terminali della grammatica sono fatti corrispondere i nodi dell'albero, e ai simboli terminali invece sono fatte corrispondere le foglie. Il lavoro del parser è quello di prendere un testo in input e riconoscere passo per passo il testo nella grammatica e nel mentre, sulla base della corrispondenza tra simboli della grammatica e nodi dell'albero, costruire l'albero di sintassi astratta.

3.6 Conclusione

Abbiamo presentato la grammatica del parser, ravvisando che è necessario rimuovere l'ambiguità. Per come si presenta il problema, è possibile eliminare l'ambiguità a livello d'implementazione del parser attribuendo a ciascuna variabile il suo tipo, in modo che ad ogni occorrenza della variabile sia chiaro a quale regola della grammatica questa appartenga.

Siccome è stata definita la struttura dell'input (ovvero la grammatica del linguaggio) e la struttura dell'output (ovvero la struttura dell'AST) possiamo procedere descrivendo l'implementazione.

```
type contract = [`Contract]
type human = [`Human]
type actor = [human | contract]
type idle_or_contract = [idle | contract]
type _ address =
 | Contract : string -> contract address
 | Human : string -> human address
type 'a tag =
 | Int : int tag
 | Bool : bool tag
 | String : string tag
 | ContractAddress : (contract address) tag
 | HumanAddress : (human address) tag
type _ tag_list =
   TNil : unit tag_list
 | TCons : 'a tag * 'b tag_list -> ('a * 'b) tag_list
type 'a ident = 'a tag * string
type ('a,'b) meth = 'a tag * 'b tag_list * string
type 'a field = 'a ident
type 'a var = 'a ident
type const = Symbolic of string | Numeric of int
type _ expr =
 | Var : 'a var -> 'a expr
 | Fail : 'a expr
 | This : (contract address) expr
 | Field : 'a field -> 'a expr
 | Plus : int expr * int expr -> int expr
 | Mult : const * int expr -> int expr
 | Minus : int expr -> int expr
 | Max : int expr * int expr -> int expr
 | Geq : int expr * int expr -> bool expr
 | Gt : int expr * int expr -> bool expr
 | Eq : 'a tag * 'a expr * 'a expr -> bool expr
 | And : bool expr * bool expr -> bool expr
 | Or : bool expr * bool expr -> bool expr
 | Not : bool expr -> bool expr
 | Value : 'a -> 'a expr
 | Symbol : string -> int expr
type _ var_list =
   VNil : unit var_list
```

```
  | VCons : 'a var * 'b var_list -> ('a * 'b) var_list
type _ expr_list =
    ENil : unit expr_list
  | ECons : 'a expr * 'b expr_list -> ('a * 'b) expr_list
type _ rhs =
  | Expr : 'a expr -> 'a rhs
  | Call : (contract address) expr option * ('a,'b) meth
     * 'b expr_list -> 'a rhs
  | CallWithValue : (contract address) expr option *
 ('a,'b) meth * 'b expr_list * int expr -> 'a rhs
type stm =
  | Assign : 'a field * 'a rhs -> stm
  | IfThenElse : bool expr * stm * stm -> stm
  | Comp : stm * stm -> stm
  | Choice : stm * stm -> stm
type ('a,'b) program = 'b var_list * stm list * 'a expr
type assign =
  | Let : 'a field * 'a -> assign
type store = assign list
type any_method_decl =
  | AnyMethodDecl : ('a,'b) meth * ('a,'b) program -> any_method_decl
type methods = any_method_decl list
type a_contract = contract address * methods * store
type a_human = human address * methods * store * human stack
```

Listing 3.1: Albero di sintassi astratta intrinsecamente tipato di scl

Capitolo 4

Parsing

4.1 Introduzione

In questa sezione ci concentriamo sull'implementazione del parser verso `scl`. Vengono quindi descritte le tecniche e le scelte implementative che portano il parser a prendere in input una lista di caratteri e determinarne la correttezza basandosi sulla grammatica, generando quindi il relativo albero di sintassi astratta per tale lista.

Il parser in considerazione è implementato in OCaml, questa scelta risulta abbastanza ovvia dal momento che lo stesso `scl` è implementato in questo linguaggio.

4.2 Analisi lessicale

È più conveniente che il parser non faccia l'analisi sintattica direttamente sul testo in input: preferiamo piuttosto che abbia in ingresso una sequenza di token; dove ogni token è una coppia nome-valore, con il nome che rappresenta una determinata categoria sintattica, mentre il valore corrisponde a una stringa del testo. Quindi per generare i token è necessario fare l'analisi lessicale, dividendo le stringhe in input in diverse categorie (le categorie sintattiche, appunto).

Con questo obiettivo usiamo il modulo `Genlex` di OCaml, che permette di generare un analizzatore lessicale che in OCaml consiste in una funzione che prende in input uno stream di caratteri e restituisce in output una lista di token. Inoltre questo strumento è particolarmente vantaggioso rispetto a un'analisi lessicale costruita da zero, perché toglie la preoccupazione di fornire un'implementazione per l'aggiunta di commenti nel testo, così come diventa automatica la rimozione degli spazi bianchi tra le varie stringhe. I token quindi sono riconosciuti e ad ognuno di essi è associato una categoria (o nome).

```
type 'ast parser =
token t -> (vartable * bool) -> token t * 'ast * (vartable * bool)
```

Listing 4.1: Dichiarazione del tipo parser

Le possibili categorie sono: interi, stringhe, identificativi e parole chiave. Mentre interi, stringhe e identificativi sono già noti al lexer, perché sono caratteristici di tutti i linguaggi, le parole chiave richiedono di essere esplicitate, perché sono specifiche del lessico del linguaggio di cui fare il parsing.

Concettualmente l'analizzatore sintattico richiederebbe uno stream di token, quindi sarebbe vantaggioso implementare il parser in modo che richieda un input del tipo `token Stream.t`. Infatti il tipo `Stream` di OCaml offrirebbe un vantaggio in termini di memorizzazione: non è necessario che tutta la sequenza di token sia in memoria, e sebbene l'analizzatore lessicale generato da `make_lexer` –la funzione di `GenLex` che lo genera– restituisca un tipo `token Stream.t`, scegliamo di usare una lista di token. Questa è una scelta di natura implementativa: come sarà più chiaro successivamente, il parser ha bisogno di fare backtracking, e con gli `Stream` di OCaml l'operazione diventerebbe ardua siccome l'eliminazione di un elemento sarebbe distruttiva. Con la lista invece l'iterazione è molto più semplice e può procedere tranquillamente in entrambe le direzioni.

L'analisi lessicale relativamente alla grammatica consiste nel riconoscere i simboli terminali come tali, in modo che successivamente ci possiamo concentrare separatamente nel riconoscimento delle produzioni della grammatica.

4.3 Implementazione con parser combinator

Una volta definita la grammatica e riconosciuti i simboli terminali di questa possiamo concentrarci sull'implementazione in OCaml del parser. Il parser può essere implementato con due diverse tecniche: similmente a quanto fatto per l'analisi lessicale, potremmo usare librerie di parsing che generano analizzatori sintattici, oppure potremmo usare la tecnica del parser combinator. In questo caso preferiamo quest'ultima tecnica, che consiste nel considerare un parser come una funzione di ordine superiore che avendo uno o più parser in input restituisce un nuovo parser output (da qui il nome). A livello di codice per parser si intende una funzione che prende una lista di token e restituisce l'albero di derivazione e la lista dei token rimanenti. Nel nostro caso il tipo `parser` coincide con quello espresso nel listato 4.1

Per l'implementazione è necessario anche definire un'eccezione che ogni funzione `parser` solleverà qualora non dovesse essere in grado di riconoscere l'input, questa eccezione è stata denominata `Fail`.

Con l'ausilio dei parser combinator possiamo tradurre facilmente le regole di produzione della grammatica: innanzitutto è necessario descrivere gli operatori sintattici necessari, ovvero la concatenazione, la stella di Kleene, l'unione, la costante e l'operatore possibilità. Nel parser l'implementazione di ogni operatore risulta semplice: a ognuno di questi facciamo corrispondere una funzione, che nella logica dei parser combinator genera un nuovo parser specifico per l'operatore in questione. Questi parser possiamo comporli tra loro, secondo le regole della grammatica, per riuscire a riconoscere una qualsiasi produzione. Mostriamo ora l'implementazione per ciascun operatore sintattico.

Per l'operatore costante –che riconosce i simboli terminali– viene generato un parser che verifica se il simbolo richiesto corrisponde al simbolo in input: nel caso in cui questo succeda l'input può essere consumato:

```
let const : token -> (token -> 'ast) -> 'ast parser =
 fun t1 f t2 tbl ->
  if (List.length t2 > 0) && (t1 = (List.hd t2)) then
   (junk t2), f t1, tbl
  else
   raise Fail
```

Per l'operatore di unione sono necessari due parser in input. Per implementare l'idea di scelta non deterministica viene eseguito il primo parser: nel caso questo sollevi `Fail` facendo backtracking viene eseguito il parser rimanente:

```
let choice : 'ast parser -> 'ast parser -> 'ast parser
= fun p1 p2 s tbl ->
 try p1 s tbl with Fail -> p2 s tbl
```

La concatenazione invece consiste nel prendere in input due parser ed eseguirli sequenzialmente unendo successivamente i due output:

```
let concat :
'ast1 parser -> 'ast2 parser ->
('ast1 -> 'ast2 -> 'ast3) -> 'ast3 parser =
fun p1 p2 f s tbl ->
  let rest1,ast1,tbl1 = p1 s tbl in
  let rest2,ast2,tbl2 = p2 rest1 tbl1 in
  rest2,f ast1 ast2,tbl2
```

La stella di Kleene vede l'esecuzione dello stesso parser finché questo non solleva `Fail` –caso per cui l'esecuzione dell'operatore termina:

```
let kleenestar :
'ast2 parser -> 'ast1 -> ('ast1 -> 'ast2 -> 'ast1) ->
```

```
'ast1 parser =
 fun p empty_ast f s t ->
  let rec aux p1 s1 acc tbl=
  try
   let (rest1, ast1, ntbl) = p1 s1 tbl in
   aux p1 rest1 (f acc ast1) ntbl
  with Fail -> (s1, acc, tbl)
  in aux p s empty_ast t
```

Per concludere, l'operatore di possibilità corrisponde un parser che semplicemente prevede che possa fallire:

```
let option : 'ast parser -> 'ast option parser =
 fun p s tbl -> try
  let next,res,ntbl = p s tbl in next,Some res,ntbl
 with Fail -> s,None,tbl
```

Dopo aver definito questi operatori diventa banale scrivere il parser. Per ogni non terminale della grammatica costruiamo una nuova funzione di tipo `parser`, costituita dalla corretta composizione delle funzioni di parsing appena descritte coerentemente con quanto definito dalle regole della grammatica. Mostriamo quindi l'esempio relativo al parsing del non terminale `ie`, dopo aver eliminato la ricorsione sinistra come abbiamo visto 3.2

```
let rec atomic_int_expr s =
 choice_list [
   comb_parser (base Int) (fun expr -> AnyExpr(Int,expr));
   concat (kwd "-") atomic_int_expr (fun _ -> minus) ;
   concat (concat (kwd "(") int_expr scd) (kwd ")") fst ;
   concat (concat (kwd "max") int_expr scd) int_expr max;
   concat (kwd "symbol") symbol_pars scd;
 ] s
and int_expr s =
 concat atomic_int_expr (option cont_int_expr)
 (fun x f -> match f with Some funct -> funct x | _ -> x) s
and binop s =
 choice_list [
  const (Kwd "+") (fun _ -> plus) ;
  const (Kwd "*") (fun _ -> mult) ;
  const (Kwd "-") (fun _ -> subtract)
 ] s
and cont_int_expr s = concat binop int_expr (fun f x -> f x) s
```

4.4 Costruzione dell'albero di sintassi astratta

Come spiegato nel capitolo precedente (5.4) tra la grammatica e l'albero di sintassi astratta per tale grammatica si ha un'associazione abbastanza diretta, quindi mentre si riconduce il testo alle produzioni della grammatica è abbastanza immediato costruire l'albero relativo. A titolo di esempio vediamo come nel caso atomico, una volta riconosciuto il token sia abbastanza immediato costruire la foglia:

```
let value : type a. a tag -> token -> a expr = fun tag tok ->
 match tag,tok with
  | String,Genlex.String x -> Value x
  | Int,Int x -> Value x
  | Bool,Kwd "true" -> Value true
  | Bool,Kwd "false" -> Value false
  | _ -> raise Fail

let value_pars tag s = const (List.hd s) (value tag) s
```

L'esempio mostra la definizione di `value_pars`, cioè il parser per il non terminale v della grammatica (tabella 3.1), nel quale la funzione `value` prende un `token` e restituisce il costrutto `Value` dell'AST `scl` (listato 3.1).

Nel caso non atomico, ovvero durante la costruzione di un nodo dell'albero, dal momento che due `parser` sono combinati per il parser è necessario fare un controllo sul tipo. È vero che essendo l'albero di sintassi astratta una *Intrinsically Typed Data Structure* non si può incorre in problemi di tipo, tuttavia al momento della costruzione della struttura è necessario fare il *type checking* in modo che i costruttori del GADT abbiano la certezza che i tipi combacino. In `scl`, ad esempio, il costruttore per l'assegnamento è definito in questo modo:

```
type stm =
  | Assign : 'a field * 'a rhs -> stm
  | ...
```

Quindi una volta che si ha un `field` e un `rhs`, per costruire il relativo `Assign` è necessario controllare che `field` e `rhs` abbiano lo stesso tipo. Sempre secondo la logica dei GADT, che permette di definire una *Intrinsically Typed Data Structure*, definiamo il tipo delle prove di uguaglianza di due tipi:

```
type (_,_) eq = Refl : ('a,'a) eq
```

Usando questo particolare tipo possiamo implementare una funzione che permetta il controllo (o meglio il cast) tra due tipi:

```
let eq_tag : type a b. a tag -> b tag -> (a,b) eq option =
fun t1 t2 ->
 match t1,t2 with
  | Int, Int -> Some Refl
  | Bool, Bool -> Some Refl
  | String, String -> Some Refl
  | ContractAddress, ContractAddress -> Some Refl
  | HumanAddress, HumanAddress -> Some Refl
  | _,_ -> None
```

Nei rami del pattern matching della funzione `eq_tag` sono istanziati i tipi in input, quindi il tipo di ritorno non solo permette di controllare se i tipi in input combacino, ma restituendo `Refl` contiene le istanze stesse dei due tipi. Una volta che abbiamo a disposizione la funzione `eq_tag` possiamo implementare un parser per l'assegnamento. Questo parser genera un `Assign` combinando il risultato dei parser che riconoscono `field` e `rhs`, e istanziando nel medesimo tipo i tipi di ritorno dei due parser:

```
let assign_pars =
 concat (concat field_pars (kwd "=") fst) rhs_pars
 (fun (AnyField(tfield,name)) (AnyRhs(trhs,r))->
 match eq_tag tfield trhs with
  | Some Refl -> Assign((tfield,name),r)
  | None -> raise Fail)
```

Nell'esempio si ha una funzione di tipo `parser` per l'assegnamento in cui, usando i parser combinator degli operatori sintattici, è esplicitata la forma sintattica (analoga alla regola di produzione nell'assegnamento nella grammatica della tabella 3.1) ed è specificato come debba avvenire la costruzione dell'albero di derivazione. In questo caso `AnyField` e `AnyRhs` sono costrutti che contengono rispettivamente il valore del campo e del *rhs* con i rispettivi tipi. Inizialmente questi due tipi sono generici, facendo quindi il pattern matching della funzione `eq_tag` su di essi possiamo ricadere nel ramo `Refl` nel quale avviene l'istanziazione dei due tipi nello stesso tipo. Dopo questa istanziazione, siccome i tipi combaciano, diventa possibile la generazione del nodo `Assign` nell'albero sintattico.

Analogamente a quanto è avvenuto in questi due casi possiamo costruire ogni nodo o foglia dell'albero.

Abbiamo quindi visto che l'utilizzo dei parser combinator ci ha portato alla generazione di un parser top-down, cioè che inizia a costruire l'albero dalla radice. Per iniziare definisce una *configuration* vuota (il simbolo iniziale) che verrà riempita dall'attività dell'analizzatore sintattico. In questo tipo di parsing l'input viene consumato da sinistra a destra con solo un simbolo di lookahead – che se non viene riconosciuto implica il

backtracking. Sebbene il parsing abbia i requisiti per essere di tipo LL(1) la grammatica in realtà non lo è. Questo perché, come visto nella sezione 3.3, presenta delle ambiguità.

4.5 Controllo dei tipi e tabella delle variabili

Le ambiguità della grammatica, stando alla sua definizione, sono dovute al fatto che in non terminali che connotano espressioni di tipo diverso può comparire la stessa variabile. Per eliminare l'ambiguità dobbiamo fare in modo che in ogni espressione, se è presente una variabile già si conosca il tipo di quest'ultima. Questo obiettivo può essere conseguito in due modi: o ad ogni occorrenza della variabile specifichiamo sia il tipo che il nome, oppure specifichiamo solo il nome della variabile ma facendo in modo che il parser possa leggere il tipo di dato in una tabella apposita (in cui è associata ad ogni nome di variabile il suo tipo). Siccome preferiamo che solo il nome sia identificativo univoco per la variabile (come avviene in quasi tutti i linguaggi di programmazione) scegliamo la seconda opzione. Di conseguenza all'interno del programma su cui effettuare il parsing non potranno mai esistere due variabili diverse ma con lo stesso tipo, cosa che in realtà è possibile nella struttura degli attori in `scl`. Così facendo si ha una perdita di espressività minima, ma si ha il vantaggio di favorire una programmazione più consapevole.

La tabella è quindi relativa a un singolo attore, e contiene la lista delle coppie tipo-nome di tutte le sue variabili. Essendo OCaml un linguaggio funzionale, non può essere che la tabella sia una variabile globale, altrimenti non potrebbe essere modificabile. È necessario quindi che la stessa funzione di tipo `parser` richieda in input e restituisca in output la tabella, in modo tale che gli sia consentito aggiungere o rimuovere elementi. In particolare una nuova variabile deve essere aggiunta alla tabella quando viene dichiarata, e quando si ha un assegnamento di una variabile non ancora presente in tabella. Ad ogni aggiunta viene controllata la tabella e nel caso fosse già presente una variabile con lo stesso nome, il parser solleva `Fail` e termina la sua esecuzione. Invece ad ogni occorrenza della variabile viene fatta sempre una ricerca nella tabella per conoscerne il tipo.

Le variabili oltre a poter essere aggiunte alla tabella devono anche poter essere rimosse. Si prenda in considerazione una variabile locale all'interno di una funzione: essa non deve essere visibile in nessun'altra funzione esterna. Quando verrà quindi fatto successivamente il parsing di un'altra funzione, se questa seconda funzione presenta una variabile con lo stesso nome questa non dovrà essere presente nella tabella. Diventa allora necessario definire lo scope per le variabili, sulla base del quale è determinata la presenza o meno della variabile nella tabella. Per come è definito il linguaggio questa è un'operazione banale, infatti in `scl` non ci sono funzioni annidate, quindi gli unici blocchi di cui può far parte una variabile sono il blocco globale dell'attore e il blocco locale della funzione. Quindi, durante il parsing, o la variabile è una variabile globale o è una variabile locale della funzione presa in esame: non è possibile avere una variabile attiva che sia allo stes-

so tempo variabile locale di un'altra funzione. Quindi basta associare ad ogni variabile nella tabella un valore booleano che denota se la variabile è locale o meno, per far sì che quando venga finito il parsing di una funzione tutte le variabili locali vengano rimosse.

Analogamente alle variabili anche le funzioni possono essere aggiunte nella stessa tabella: quando viene dichiarato un metodo, la *signature* di questo –nome, lista dei tipi dei parametri e tipo di ritorno– viene salvata nella tabella dell'attore. Così, ogni volta che ci sarà una chiamata di funzione, dal nome di questa viene cercata la *signature* del metodo nella tabella, e si verifica che la lista delle espressioni nella chiamata sia coerente con la lista dei parametri presa dalla *signature* appena ritornata. In `scl` però si possono chiamare i metodi di uno specifico contratto, in questo caso scegliamo che non verrà fatto alcun controllo perché il contratto potrebbe essere esterno al testo su cui fare il parsing e, in questo caso, non si avrebbe alcuna informazione.

Un'altra informazione che il parser deve conoscere durante la sua esecuzione è se l'attore che sta analizzando sia umano o contratto: come abbiamo visto prima, all'umano sono permessi più comandi e quindi il suo stesso codice può presentare diversi comandi. Questa informazione, che si traduce in un booleano, la affianchiamo alla tabella come input e output di ogni funzione di tipo `parser`.

4.6 Conclusione

Abbiamo quindi visto che per implementare il parser LL(1) abbiamo dovuto definire in primo luogo una sintassi per `scl`. Con la tecnica dei parser combinator ci è poi bastato trascrivere le stesse regole della grammatica come combinazione di più parser, facendo però alcuni accorgimenti: abbiamo tolto la ricorsione sinistra per evitare che il parser divergesse. Grazie all'uso dei GADT, a livello di implementazione del parser, abbiamo fatto in modo che il parsing si concluda con successo solamente se c'è concordanza tra i tipi effettivamente usati nelle espressioni.

Abbiamo, invece, aggirato il problema dell'ambiguità della grammatica mediante la creazione di una tabella che permette di ricordare l'associazione tra nome di variabile e tipo per tutta la durata del parsing.

Capitolo 5

Compilazione

5.1 Introduzione

Una volta che si hanno a disposizione tutti gli strumenti per programmare in `scl` ciò che viene a mancare è un'esecuzione vera e propria del codice su una blockchain: `scl` fornisce tutti i mezzi per fare analisi, ma solo con `scl` non è possibile fare il deploy di ciò che si scrive.

In questo capitolo mostriamo l'implementazione di un compilatore da `scl` verso un linguaggio per *smart contract* già conosciuto, ovvero Solidity. Questa scelta è dovuta, principalmente al fatto che è il più conosciuto linguaggio per *smart contract* e, a differenza di altri (come ad esempio Liquidity), allo stato attuale è difficile che venga abbandonato. Con Solidity diventa possibile fare il deploy dei contratti sulla sua blockchain Ethereum.

5.2 Solidity ed Ethereum

Ethereum è una piattaforma decentralizzata che permette lo sviluppo di *smart contract* da parte di programmatori mettendo a disposizione Solidity: un linguaggio che in una logica di programmazione ad oggetti class-based permette di programmare *smart contract*. Il compilatore di Solidity traduce in un primo momento il programma in bytecode per poi eseguirlo nella EVM, la macchina virtuale di Ethereum – la quale costituisce, quindi, l'ambiente di esecuzione degli *smart contract*.

Solidity è un linguaggio imperativo che permette di definire delle classi di contratti, che come nella logica dei linguaggi orientati agli oggetti possono essere ereditate da altre classi o interfacce, specificando metodi e campi. Ogni contratto nella blockchain ha un suo indirizzo specifico a cui altri contratti si possono riferirsi. Inoltre tutti i contratti

hanno un campo specifico `balance` che consiste nella quantità di Ether (criptovaluta di Ethereum) posseduti dal contratto, quantità che può essere modificata con transazioni (chiamate di funzioni) da parte di utenti o contratti. Le funzioni possono essere marcate come `payable`, in modo tale da poter ricevere Ether quando chiamate. Ci sono poi indirizzi speciali come il noto `this` e `msg.sender`, dove quest'ultimo si riferisce all'indirizzo al chiamato. L'interfaccia di un contratto è rappresentata dal Abstract Binary Interface (*ABI*), nel quale sono contenuti le *signature* dei metodi e i campi. L'*ABI* diventa interessante nella fase di deploy, in quanto con l'indirizzo del contratto costituisce l'insieme delle informazioni necessarie per la creazione del contratto stesso. Solidity permette di riferirsi a un indirizzo di un contratto tramite la parola chiave `address`, che costituisce il tipo generico di un oggetto (analogamente a `Object` in Java), e che diventa utile in fase di compilazione nell'eventualità in cui si debba riferire a un contratto non conosciuto. Ogni contratto può definire una funzione di `fallback`, che viene eseguita quando dall'esterno ci si riferisce a una funzione non presente nell'*ABI* del contratto stesso.

Solidity quindi definisce solo i contratti, non mette a disposizione gli strumenti per definire gli utenti: infatti chi vuole interagire coi contratti di Ethereum deve iscriversi al servizio. Quindi nella compilazione che si va ad esporre verrà tralasciato l'aspetto relativo agli utenti.

5.3 Differenze tecniche tra scl e Solidity

Abbiamo appena introdotto una differenza importante tra `scl` e Solidity: il primo linguaggio definisce un *oggetto* contratto (ovvero una singola istanza), mentre il secondo definisce la *classe* del contratto. È necessario marcare questa differenza perché in `scl` nella configurazione di un contratto si possono fare riferimenti specifici ad altri contratti. In aggiunta a questo nella configurazione non c'è un contratto principale, ma tutti i contratti sono "sullo stesso piano": è necessario mantenere una situazione analoga in Solidity. Non possiamo quindi delegare a un singolo contratto il compito di creare gli altri. Conseguentemente i riferimenti specifici agli altri contratti della configurazione non possono essere creati all'interno di una classe. È quindi necessaria una fase di deploy in cui tutti i contratti verranno creati sulla blockchain, fase che verrà approfondita nel capito 6. Al momento della creazione ad ogni contratto è assegnato un indirizzo, a cui gli altri contratti potranno fare riferimento.

5.4 Albero di sintassi astratta per Solidity

Essendo `scl` un codice minimale, nella fase di compilazione si genera un sottoinsieme del codice di Solidity: non saranno catturati tutti i suoi costrutti, ma solo quelli di cui

ne fa relativo uso `scl`. Nella prima fase del processo di compilazione abbiamo tradotto un AST di `scl` in un AST che fedele ai costrutti di Solidity.

L'albero di sintassi astratta generato dalla compilazione è sempre intrinsecamente tipato, quindi l'obiettivo è rappresentare la struttura di un generico programma ben tipato in Solidity. Il compilatore quindi deve cercare di tradurre un AST intrinsecamente tipato di `scl` in un AST intrinsecamente tipato per Solidity. Siccome ad un programma ben tipato in `scl` deve corrispondere uno ben tipato di Solidity, dobbiamo creare tra i due AST una corrispondenza di tipi. Una volta creata questa corrispondenza ad ogni costrutto `scl` deve essere fatto corrispondere il medesimo in Solidity. Come abbiamo visto, i costrutti `scl` sono dei costrutti molto comuni ai linguaggi di programmazione (*if then else*, chiamate e definizioni di funzioni, operatori algebrici e logici): la traduzione verso Solidity di questi costrutti risulta abbastanza banale.

Come visto nel capitolo 3.3 i tipi in `scl` sono gli interi, i booleani, le stringhe, i contratti e gli umani. I primi tre di questi tipi sono dei tipi primitivi in Solidity e quindi ad ognuno di questi può essere fatto corrispondere il relativo tipo. Gli umani invece non sono presenti in Solidity, ma siccome la compilazione consiste nella traduzione solo dei contratti il compilatore può sollevare un'eccezione quando incorre in una traduzione di un umano. Bisogna invece prestare più attenzione nella traduzione del tipo contratto. Ovviamente ad ogni contratto definito in `scl` è fatta corrispondere una classe di quel contratto, ma come definire gli oggetti contratto all'interno di una classe è una questione più delicata: in `scl` i contratti non si distinguono tra loro, perché mancano di interfaccia.

Sebbene non presenti grandi differenze rispetto all'AST per `scl` nella definizione dell'AST per Solidity vengono aggiunti nuovi nodi. Nel listato 5.1 notiamo che nella definizione di funzione (`meth`) viene specificato oltre al nome, i parametri e il tipo di ritorno, che erano già presenti in `scl`, la visibilità e la view – che sono caratteristiche di Solidity. La visibilità, come in altri linguaggi di programmazione ad oggetti, denota chi può chiamare la funzione, mentre la view (più specifica per Solidity) è relativa alla possibilità che ha la funzione di cambiare lo stato interno. Come preannunciato la maggior modifica è relativa ai tipi contratto: non si ha più un unico `Contract`, ma si ha sia il tipo `Interf` che il tipo `Address` – e due nuove espressioni `CastInterf` e `Addr`, le quali permettono il cast tra i due tipi. Questa distinzione di tipi è dovuta al fatto che, come esposto prima, in Solidity sono realizzate le classi relative ai contratti `scl`, ma un contratto con gli altri contratti della configurazione interagisce mediante l'indirizzo.

```
type 'a typename =
  | Int : int typename
  | String : string typename
  | Bool : bool typename
  | Interf : interface_id typename
  | Address : addr typename
and addr = AddrInt of interface_id
and interface_id = InterfaceId of string
and storage = Storage | Memory | Calldata
and 'a var = 'a typename * string
and 'a expression =
  | Var : 'a var -> 'a expression
  | Value : 'a -> 'a expression
  | MsgValue : int expression
  | Balance : int expression
  | This : interface_id expression
  | CastInterf : interface_id * addr expression ->
     interface_id expression
  | Addr : interface_id expression -> addr expression
  | Plus : int expression * int expression -> int expression
  | Minus : int expression * int expression -> int expression
  | Mult : int expression * int expression -> int expression
  | Symbol : string -> int expression
  | Geq : int expression * int expression -> bool expression
  | Gt : int expression * int expression -> bool expression
  | Eq : 'a typename * 'a expression * 'a expression ->
     bool expression
  | And : bool expression * bool expression -> bool expression
  | Or : bool expression * bool expression -> bool expression
  | Not : bool expression -> bool expression
  | CondExpr : bool expression * 'a expression * 'a expression ->
     'a expression
  | Call : interface_id expression * ('a, 'b) funct *
 'a expression_list * (int expression) option-> 'b expression
and _ expression_list =
    ExprNil : unit expression_list
  | ExprCons : ('a typename * 'a expression) * 'b expression_list ->
     ('a * 'b) expression_list
and _ param_list =
    PNil: unit param_list
```

```
  | PCons: ('a var * storage option) * 'b param_list ->
      ('a * 'b) param_list
and view = View | Pure | Payable
and visibility = External | Public | Internal | Private
and ('a, 'b) funct = string * 'a param_list *
          ('b typename * storage option) option *
          view list * visibility list
and meth = Funct : (('a, 'b) funct) -> meth
and interface = Interface : (interface_id * meth list) ->
interface
and declaration = Declaration : 'a var * ('a expression option) ->
    declaration
and statement =
  | Empty
  | IfElse : bool expression * statement * statement -> statement
  | Assignment : 'a var * 'a expression -> statement
  | Sequence : statement * statement -> statement
type any_funct=  Function: ('a, 'b) funct * statement *
    ('b expression option)-> any_funct
type contract_ast = Contract of string * (declaration list) *
(any_funct list) * int
```

Listing 5.1: Albero di sintassi astratta intrinsecamente tipato per Solidity

Una volta generato l'AST intrinsecamente tipato per Solidity, ci dobbiamo preoccupare solamente di tradurre l'albero effettivo codice Solidity. Dal momento che l'AST riprende fedelmente i costrutti del linguaggio, questa operazione diventa piuttosto banale, poiché consiste semplicemente nell'assegnare ad ogni costrutto la giusta stringa.

5.5 Interfacce per le variabili Contract

Dobbiamo, quindi, riprodurre in Solidity una configurazione in cui tutti i contratti si conoscono: per cui questi contratti devono poter essere espressi come campi all'interno di un contratto. Abbiamo visto che in `scl` esiste il tipo `Contract`, di conseguenza è possibile avere variabili di tipo contratto. Il problema diventa, quindi, come esprimere queste variabili in Solidity, e in particolare decidere quale possa essere il loro tipo. Logicamente se volessimo rimanere aderenti al codice `scl` queste dovrebbero essere di tipo `address`. Il problema però è che se ci si riferisse solo all'indirizzo del contratto allora anche le chiamate di funzione dovrebbero essere fatte sull'indirizzo –sarebbe come fare in Java una chiamata di metodo su un oggetto di tipo `Object`. Solidity permette questo tipo di chiamate a basso livello usando la funzione `call`; tuttavia il loro tipo di ritorno è

espresso in byte e per fare la conversione saremmo obbligati a scrivere codice Assembly. Preferiamo, quindi, un'altra tecnica, dove le chiamate possano essere fatte su degli oggetti meno generici e dove si conosca il tipo di ritorno delle funzioni.

Ad ogni campo di tipo contratto allora facciamo corrispondere un'interfaccia che contiene la *signature* dei suoi metodi. Solidity permette, come già detto, di fare il cast da interfaccia a indirizzo e viceversa. In questo modo è necessario lavorare sia con gli indirizzi dei contratti, riuscendo a rimanere fedeli al codice `scl`, e allo stesso tempo usare le interfacce, necessarie per le chiamate di funzione. In particolare scegliamo che i campi (sempre di tipo contratto) siano oggetti che abbiano come tipo la loro interfaccia, e che i parametri siano indirizzi. Nel momento della loro dichiarazione assoceremo al parametro la sua relativa interfaccia. Conseguentemente, se si ha una chiamata di funzione su un parametro, di questo verrà fatto il cast alla sua interfaccia; mentre se viene passato un oggetto di tipo interfaccia come parametro attuale di una funzione, di questo viene fatto il cast verso l'indirizzo. È necessario che ad ogni chiamata di un metodo di un contratto, venga aggiunta la *signature* del metodo nell'interfaccia.

```
Contract sample {
    Contract interf
    int x
    function m (Contract addr): int{
        x = addr.f(interf)
        x = interf.g(addr)
        return x
    }

}
```

Nell'esempio di codice `scl` vediamo un campo di tipo contratto (`interf`) e un parametro di tipo contratto (`addr`), la compilazione verso Solidity diventa:

```
interface Interf0{
    function g(address) external returns (int);
}
interface Interf1{
    function f(address) external returns (int);
}
contract sample {
    Interf0 interf;
    int x;

    function m(address addr) public returns (int){
        x = Interf1(addr).f(address(interf));
```

```
        x = interf.g(addr);
        return x;
    }
}
```

Notiamo quindi che `interf` è tradotto come un oggetto con la sua interfaccia `Interf0`, mentre `addr`, che nel codice `scl` è sempre del tipo `Contract`, è un oggetto generico di tipo `address`– ma nonostante ciò ad `addr` viene associata l'interfaccia `Interf1`. Quando viene chiamata una funzione su `addr`, viene prima fatto il cast in `Interf1`, mentre quando viene chiamata una funzione su `interf` non viene fatto nessun cast. Invece, nel passaggio di parametri, si ha il cast verso `address` quando viene passata `interf`, mentre non si ha il cast di `addr` siccome è già del tipo `address`. Notiamo che le due funzioni chiamate sono aggiunte nelle relative interfacce: la funzione `g`, chiamata da `interf` è aggiunta in `Interf0`, e la funzione `f` chiamata da `addr` è aggiunta in `Interf1`.

Rimane da tradurre la situazione in cui si ha un assegnamento di contratti con interfacce diverse. In pratica si ha che deve avvenire un cast verso l'interfaccia del *lhs*, però, dal momento che i tipi in Solidity sono nominali, sarebbe necessario che la classe del *lhs* implementasse esplicitamente l'interfaccia del *rhs*. Si potrebbe quindi aggiungere manualmente i metodi all'interfaccia del *lhs*, o comunque far ereditare il *lhs* al *rhs*. Aggiriamo però il problema in modo che Solidity se ne preoccupi a tempo di esecuzione, invece che durante la compilazione. Infatti facciamo prima un cast dal *rhs* al suo `address` e poi dall'indirizzo all'interfaccia. In questo modo è possibile fare qualsiasi cast tra due diversi oggetti senza che il compilatore Solidity sollevi un'eccezione.

```
Contract p
Contract q
p = q
```

L'assegnamento diventa:

```
Interf0 p;
Interf1 q;

p = Interf0(address(q));
```

5.6 Inizializzazione degli indirizzi

Definito quindi come avviene la traduzione per i tipi `Contract` di `scl`, non è ancora chiaro però come il compilatore possa far conoscere ad ogni contratto i riferimenti degli altri contratti della configurazione. Ovviamente in Solidity sarebbe necessario che un contratto debba avere gli indirizzi degli altri contratti. Questi indirizzi non possono

essere noti a tempo di compilazione poiché sono indirizzi relativi al posizionamento del contratto nella blockchain e quindi sono assegnati al momento del deploy. Per fare in modo che a tempo di deploy si possano comunicare al contratto tutti gli indirizzi richiesti è necessario che ogni contratto abbia un campo per ogni altro contratto della configurazione. Il compilatore, poi, crea una funzione `init` che assegna gli indirizzi passati come parametro ai rispettivi campi. Questa funzione verrà poi chiamata in fase di deploy quando saranno noti gli indirizzi effettivi.

Questa configurazione base in `scl`

```
Contract a{}
Contract b{}
Contract c{}
```

Compilata in Solidity diventa:

```
contract a {
    Interf1 c;
    Interf0 b;
    bool initialize = false;

    function init(address _b, address _c) public {
        if (!initialize){
            b = Interf0(_b);
            c = Interf1(_c);
            initialize = true;
        }
    }
}
contract b {
    Interf3 c;
    Interf2 a;
    bool initialize = false;

    function init(address _a, address _c) public {
        if (!initialize){
            a = Interf2(_a);
            c = Interf3(_c);
            initialize = true;
        }
    }
}
contract c {
```

```
    Interf5 b;
    Interf4 a;
    bool initialize = false;

    function init(address _a, address _b) public {
        if (!initialize){
            a = Interf4(_a);
            b = Interf5(_b);
            initialize = true;
        }
    }
}
```

dove la variabile booleana `initialize` assicura che la funzione `init` non venga chiamata più volte. In questo modo evitiamo che un attaccante possa cambiare l'indirizzo di un contratto noto in un indirizzo di un contratto terzo. Così ogni contratto si può riferire agli altri contratti della configurazione.

Durante il deploy siamo anche interessati a inizializzare i `balance` dei contratti. Per fare questo è necessario che il compilatore generi il costruttore per ogni contratto e lo marchi come `payable`, in questo modo quando inizializzeremo il contratto basterà chiamare il costruttore e passargli il giusto quantitativo di Ether.

5.7 Considerazioni implementative

5.7.1 Associazione tra i tipi di scl e Solidity

A livello implementativo, la traduzione da AST di `scl` a AST di Solidity è abbastanza semplice. Partiamo creando un'associazione tra i tipi di dato in `scl` e tipi in Solidity. Questi abbiamo visto essere gli stessi ad eccezione degli umani, che non sono presenti in Solidity, e dei contratti di `scl` che vengono tradotti in indirizzi o oggetti di tipo interfaccia. Abbiamo quindi un'associazione abbastanza semplice:

```
let get_typename  : type a b. a typename -> b tag -> any_typename
= fun typ tag ->
  match tag with
  | SmartCalculus.Int -> Typename Int
  | SmartCalculus.Bool -> Typename Bool
  | SmartCalculus.String -> Typename String
  | SmartCalculus.ContractAddress -> Typename typ
  | SmartCalculus.HumanAddress -> raise CompilationFail
```

Pertanto se il compilatore cercasse di tradurre il tipo umano verrebbe sollevata un'eccezione, mentre per tradurre il contratto, a seconda del contesto, dovrebbe specificare il relativo tipo di ritorno (o indirizzo o interfaccia). In tutti gli altri casi il tipo rimarrebbe il medesimo.

Una volta creata questa associazione, la costruzione dell'albero è banale in quanto quasi tutti i costrutti `scl` sono già presenti in Solidity. In più grazie all'implementazione tramite GADT dei due alberi non è necessario nessun vero *type checking*: sappiamo che l'albero `scl` è già ben tipato e di conseguenza, se viene rispettata, l'associazione anche l'AST Solidity sarà ben tipato. Ci rimane solo da implementare i costrutti di `scl` non presenti in Solidity, in particolare `symbol`, e mettere in piedi il meccanismo per associare ad ogni contratto la sua interfaccia.

5.7.2 Implementazione del costrutto symbol

Il costrutto `symbol` di `scl` permette di assegnare una stringa a un intero. Il razionale è che, se ci si vuole riferire a una costante senza interesse verso l'effettivo valore, è più comodo usare una stringa piuttosto che un intero. Ovviamente Solidity non mette a disposizione alcuna funzione che implementa symbol. Possiamo però usare il tipo `mapping` di Solidity, che rende possibile definire delle funzioni hash. In particolare possiamo una funzione hash `symbol` che associ a una stringa il relativo intero. Per implementare questa funzione, però, è necessario che a stringhe già definite sia associato lo stesso valore – altrimenti si perderebbe la definizione di funzione hash. Il compilatore pertanto, durante la costruzione dell'albero, deve conoscere quali stringhe di `symbol` sono già state utilizzate. Similmente a quanto avvenuto nell'implementazione del parser, anche per l'implementazione del compilatore è necessaria una tabella a cui il compilatore fa riferimento. In questa tabella devono essere contenuti tutte le stringhe di `symbol` già conosciute dal compilatore, così quando si ha la traduzione di questo costrutto la relativa stringa viene aggiunta alla tabella solo nel caso in cui non sia ancora presente. Quando poi verrà tradotto l'AST in codice Solidity è necessario che nel costruttore venga associato a ogni `symbol` un valore diverso. Mostriamo quanto detto nell'esempio:

```
Contract a{
    function foo() : int{
        x = symbol("something")
        x = symbol("else")
        x = symbol("something")
        return x
    }
}
```

Il contratto `a` verrà quindi compilato in

```
contract a {
    mapping (string => int) symbol;
    int x = 0;
    constructor() payable public {
        symbol['else'] = 0;
        symbol['something'] = 1;

    }
    function foo() payable public returns (int ){
        x = symbol['something'];
        x = symbol['else'];
        x = symbol['something'];
        return x;
    }
}
```

Abbiamo quindi che vengono aggiunte alla funziona hash `symbol` solo due stringhe diverse `"something"` ed `"else"`, a cui vengono fatti corrispondere due valori interi diversi, rispettivamente `1` e `0`.

5.7.3 Implementazione delle interfacce

In `scl` è possibile che si abbia un riferimento a una variabile senza che questa sia dichiarata, ma questo non può avvenire in Solidity. Ancora una volta il compilatore necessita di una tabella nella quale all'occorrenza aggiunga i campi con il relativo tipo (esattamente come avveniva in `scl`). In questo modo quando viene fatta la traduzione in codice Solidity si aggiungono nelle dichiarazioni tutti i campi in precedenza non dichiarati. Questa tabella è inoltre utile per implementare le interfacce di cui precedentemente si è discusso.

Abbiamo visto infatti che ogni contratto – inteso come variabile oggetto nella classe – ha bisogno di essere associato a un'interfaccia. Anche in questo caso è necessario che il compilatore conosca questa associazione, in modo da poter modificare, aggiungendo metodi, le interfacce già presenti. Questa associazione è mantenuta all'interno della tabella delle variabili: ad ogni variabile di tipo contratto il compilatore aggiunge un valore con l'id (che sarebbe il nome) dell'interfaccia relativa. Perciò è necessaria al compilatore anche un'altra tabella in cui ad ogni interfaccia è associata la lista dei suoi metodi. Per cui, quando si ha una chiamata di funzione su un contratto (e quindi il compilatore deve aggiungere la funzione all'interfaccia del contratto), viene prima cercato l'id dell'interfaccia nella tabella delle variabile, poi dall'id si trova la relativa interfaccia nella tabella che contiene tutte le interfacce: non rimane che aggiungere a questa il metodo.

5.8 Problemi con il sistema di tipi in Solidity

Solidity è un linguaggio tipato staticamente, e ciò che ci si aspetterebbe, quindi, è che il compilatore Solidity garantisca la coerenza tra i tipi, e nel caso non vengano rispettati dei vincoli di tipo non generi il bytecode necessario per l'esecuzione. Durante l'implementazione della compilazione, però, ci siamo accorti di come sia semplice aggirare il controllo statico dei tipi del compilatore. Mostriamo adesso un esempio in cui il compilatore sollevi giustamente un errore di tipo:

```
contract c1 {
    function foo() payable external returns (bool){
        return true;
    }
}

contract c2 {
    function fie(bool) payable external returns (int){
        return 42;
    }
}

contract sample{

    function test(int, bool) payable public{
        c2 b = new c2();
        b.foo();
    }
}
```

In questo caso nella funzione `test` di `sample` il contratto `b` sta facendo una chiamata a un metodo di un'altra classe: il compilatore correttamente solleva un errore di tipo. Facendo però una semplice modifica solamente alla funzione `test` il compilatore non solleverà più l'eccezione:

```
contract sample{

    function test(int, bool) payable public{
        c2 b = new c2();
        c1(address(b)).foo();
    }
}
```

Ci si chiede allora come viene fatto il cast da `c1` a `c2` visto che sono due classi di contratti con metodi diversi. Il cast di Solidity in realtà ha solo valenza sintattica: in questo caso non si riscontra nessun errore a tempo di compilazione, ma verrà sollevato un errore a tempo di esecuzione, quando si cercherà di risolvere la chiamata.

Questa mancanza di un controllo statico del cast, può risultare particolarmente dannoso. Un pattern spesso usato nel linguaggio Solidity è `msg.sender.transfer(n)` che consiste nel trasferire `n` Ether all'indirizzo del chiamante, passando per la sua speciale funzione *fallback*. Se questa non dovesse essere definita si avrebbe un errore a tempo di esecuzione, che comporterebbe un mancato trasferimento di denaro – come ulteriormente approfondito nell'articolo (Crafa and Pirro 2019).

5.9 Conclusione

In questo capitolo abbiamo mostrato come avviene la compilazione verso Solidity per un programma in `scl`. Viene quindi prima fatta la traduzione da AST `scl` a AST Solidity, e da quest'ultimo viene generato il codice. Questi due procedimenti sono abbastanza banali, se non per il fatto che in Solidity vengono definiti le classi dei contratti, diversamente a quanto avviene in `scl` dove vengono definiti gli oggetti. È necessario inoltre poter fare le chiamate di funzioni sugli oggetti di tipo contratto, e questo lo abbiamo reso possibile assegnando ad ogni tipo una sua interfaccia. Durante la fase di implementazione del compilatore abbiamo anche riscontrato dei limiti del sistema di tipi statico del linguaggio Solidity, dove si ha che nel cast non avviene nessun controllo sul tipo: serve solo a non far sollevare un'eccezione al compilatore.

Capitolo 6

Deploy

6.1 Introduzione

Una volta che abbiamo il codice Solidity mancano solo gli effettivi contratti sulla blockchain. Per fare deploy è necessario uno script che si colleghi alla rete e che crei l'istanza dei contratti. Come linguaggio di scripting è stato scelto Python perché dispone dei mezzi necessari per interagire con Ethereum. Per la fase di testing, invece di creare i contratti direttamente sulla rete di Ethereum, è stata creata una rete locale.

6.2 Implementazione script Python

Il compilatore, oltre al codice Solidity, deve generare il codice dello script Python. Questo codice si deve preoccupare quindi di creare tutti i contratti della configurazione, avendo così per ciascuno il proprio *ABI* e il proprio indirizzo. Per ogni contratto sono necessarie 3 cose: in primo luogo serve che gli venga assegnato il `balance` iniziale, poi che venga chiamata la funzione `init` definita nel capitolo 5.6 (a cui vengono passati come parametri gli indirizzi degli altri contratti), e per ultimo che vengano salvati l'indirizzo e l'ABI, che costituiscono gli elementi necessari per potersi riferire al contratto.

Per la creazione dei contratti, lo script deve prima compilare il codice Solidity generato in precedenza, questo è reso possibile dalla funzione `compile_source` della libreria `solc` che compila il file Solidity passato per parametro: in particolare crea una lista dove all'interno di ogni elemento è presente l'id e l'interfaccia di ogni contratto.

La libreria invece che permette di interagire con Ethereum invece è `Web3.py`. Questa mette a disposizione metodi per collegarsi a un nodo di Ethereum e permette di generare contratti nella blockchain coi quali interagire. Dall'interfaccia del contratto generata

dopo la compilazione è possibile estrapolare l'ABI e il bytecode relativi ai contratti, che sono gli elementi necessari per costruire il contratto. Una volta che si ha l'oggetto contratto è possibile aggiungere il `balance` iniziale relativo passandolo come `value` al costruttore. Una volta chiamato il costruttore il contratto viene aggiunto nella blockchain e gli viene assegnato un indirizzo:

```
def deploy_contract(w3, contract_interface, balance):
    contract = w3.eth.contract(
        abi=contract_interface['abi'],
        bytecode=contract_interface['bin'])
    tx_hash = contract.constructor().transact({'value': balance})
    print('waiting for address...')
    address = w3.eth.waitForTransactionReceipt(tx_hash)
    return address
```

Dopo che un contratto è stato creato, è possibile chiamare qualsiasi funzione del contratto con visibilità `external` o `public`; quindi è possibile anche impostare gli indirizzi chiamando la funzione `init` del contratto. Lo script deve comunicare anche tutte le informazioni necessarie per rendere possibile nuovamente il deploy della stessa configurazione in un secondo momento. Sostanzialmente per ogni oggetto contratto deve essere comunicato l'indirizzo e l'ABI, in modo da poter ricreare il medesimo contratto: il primo consiste in una stringa che lo script stampa, mentre il secondo è un dizionario in Python che viene salvato come file JSON – e di detto file viene stampato il percorso.

6.3 Conclusione

Abbiamo visto che il compilatore, oltre a generare il codice Solidity che definisce i contratti in modo conforme alla struttura `scl`, genera codice Python che, quando viene eseguito, fa l'effettivo deploy dei contratti sulla blockchain Ethereum costruita in una rete locale. Tale codice oltre a fare il deploy, inizializza i contratti con il relativo `balance` e, tramite la funzione `init`, permette che ogni contratto conosca effettivamente gli indirizzi sulla blockchain di tutti gli altri contratti della configurazione. Una volta eseguito questo codice abbiamo tutti gli strumenti per fare il deploy al fine di rendere effettivi i contratti creati in `scl` e poter condurre analisi su questi.

Capitolo 7

Conclusione

In questo lavoro abbiamo esaminato il processo che ha portato alla realizzazione di un'infrastruttura che permette di eseguire completamente le istanze di *smart contract* definite dal linguaggio `scl` – che in precedenza era solo un linguaggio teorico su cui fare analisi.

Questo è stato reso possibile in un primo momento assegnando al linguaggio `scl` una sintassi e implementando il relativo parser, in modo da rendere più semplice la stesura del codice. Abbiamo visto che questo parser è LL(1): quindi, operando in maniera ricorsiva, dalla radice costruisce (da sinistra a destra) un albero di derivazione per il testo in input nella grammatica definita. In aggiunta è stato implementato un compilatore da `scl` a Solidity, che per ogni contratto `scl` crea la relativa classe. Per riprodurre una configurazione `scl` (caratterizzata dal fatto che tutti i contratti conoscono i riferimenti agli altri contratti della configurazione), abbiamo fatto in modo che in Solidity ogni classe relativa a un contratto contenesse i campi con i riferimenti necessari. Per creare le istanze relative alle classi di Solidity abbiamo fatto generare al compilatore un codice Python che – su una rete di testing locale – istanziasse i contratti definiti in Solidity e comunicasse ad ogni contratto i riferimenti necessari, rendendo possibile il loro deploy sulla blockchain.

La realizzazione di questa infrastruttura ha dato luogo a diversi argomenti di interesse. Prima di tutto abbiamo visto come i GADT di OCaml permettano di implementare una *Intrinsically Typed Data Structure* – e, nel contesto del parser e del compilatore, abbiamo visto come questa eviti di sottoporre gli AST a ridondanti controlli di tipo. Abbiamo avuto modo di sperimentare anche la tecnica dei parser combinator, osservando come questa consenta di definire in modo modulare il comportamento del parser nelle notazioni della grammatica. Inoltre, durante la fase di implementazione abbiamo reso noti i problemi riscontrati col sistema di tipi di Solidity: sebbene sia un linguaggio con

controllo statico dei vincoli di tipo, abbiamo mostrato quanto sia facile scrivere del codice mal tipato in cui il compilatore non rilevi errori di tipo.

In termini di tempo il mio lavoro ha richiesto circa 400 ore. Dal punto di vista della quantità di codice l'implementazione del parser ha richiesto circa 600 righe, buona parte necessarie per dover esprimere ogni costrutto `scl` tramite il parsing e molte altre per che hanno a che vedere con la gestione della tabella di parsing. Il compilatore invece ha richiesto circa 800 righe di codice: una prima parte, costituita all'incirca da un centinaio di righe, per la definizione dell'albero di sintassi astratta; le restanti equamente divise tra il codice per la generazione dell'AST per Solidity e il codice per permettere la traduzione in linguaggio Solidity di ogni ramo dell'AST. A queste vanno aggiunte un altro centinaio di righe di codice necessarie per la generazione del codice Python per il deploy.

7.1 Approfondimenti futuri

Il lavoro svolto lascia spazio a diversi approfondimenti futuri. In primo luogo, per avere una compilazione più completa per `scl`, oltre a realizzare la compilazione per gli attori contratto sarebbe necessario realizzare anche la compilazione relativa agli attori umani. Ovviamente questa non può avvenire verso Solidity – che non permette di definire gli utenti – ma dovrebbe essere verso un linguaggio che permetta di interagire con i contratti sulla blockchain (un tale linguaggio, ad esempio, potrebbe essere Python). È inoltre possibile che in futuro si sviluppino diverse analisi, sempre relative agli *smart contract*, che potrebbero partire da linguaggi diversi da `scl`, ma con simili caratteristiche. Un'altra possibile evoluzione del mio lavoro potrebbe concretizzarsi nell'utilizzo dei tool che ho implementato per scl e nel riadattamento di questi a nuovi linguaggi.

Ringraziamenti

In primo luogo ringrazio il professor Coen per avermi dato l'opportunità di realizzare una tesi che suscitasse il mio interesse, per la sua generosa disponibilità mostrata nel risolvere ogni mio dubbio o chiarimento, e per avermi introdotto alla programmazione funzionale: argomento che mi ha sempre affascinato ma che non avevo mai avuto occasione di studiare.

Ringrazio i miei genitori che, supportando i miei studi, mi hanno dato la possibilità di intraprendere questa strada. Li ringrazio anche per la comprensione e la pazienza mostrata ultimamente, il fatto che da più di un mese io abbia smesso di stirare ne è la prova. Ringrazio i miei fratelli, perché (oltre a non essersi lamentati del fatto che non stiri più) la loro presenza in casa è sempre qualcosa di speciale.

Ringrazio i membri della Fratellanza che coerentemente al quarto articolo che dice *"La Fratellanza occorre sempre in aiuto dei suoi pirati se questi in difficoltà"* mi hanno dato tutto il loro sostegno. In primis ringrazio Monica, per lei potrei fare una lista infinita ma in particolare la ringrazio per la pazienza che in questo periodo ha mostrato nei miei confronti: nonostante il mio essere scorbutico, standomi sempre accanto è stata la principale fonte di distrazione dalle mie ansie e preoccupazioni, in lei ho veramente trovato l'affetto di cui avevo bisogno. Ringrazio poi Emanuele, al quale, dopo anni e anni, mi posso sempre affidare per qualsiasi cosa, in particolare in quest'ultimo periodo in cui ogni suo aiuto è stato prezioso. Sebbene l'Informatica lo consideri un traditore lui le ha dato un piccolo contributo, che si trova nelle pagine di questa tesi. Ringrazio Fabio (e le sue ventitré personalità) perché la sua serenità e la sua spontaneità lasciano sempre un sorriso sul mio viso.

Ringrazio anche i miei amici universitari, che in questi tre anni di gioie e dolori mi sono spesso venuti in soccorso. Dagli esami preparati insieme ai posti tenuti a lezione, dai babbi natali segreti al pattinaggio in cui c'è sempre qualcuno che pacca, loro hanno contribuito a farmi raggiungere questo traguardo. Un ringraziamento speciale va a Giulia, per avermi sempre dato una mano nelle "cose burocratiche", e per avermi prestato i suoi appunti (non esiste criptovaluta in grado di stimarne l'effettivo valore) prima di ogni esame.

Infine ringrazio tutti gli amici e parenti (non li elenco tutti per motivi di tempo) per il sostegno e la forza che mi danno ogni giorno.

Bibliografia

Crafa, Silvia, and Matteo Di Pirro. 2019. "Solidity 0.5: When Typed Does Not Mean Type Safe." *CoRR* abs/1907.02952. http://arxiv.org/abs/1907.02952.

Ethereum. 2018. "Solidity Docs." https://solidity.readthedocs.io/en/v0.5.13/.

Ke, Junjie. 2017. "Applications of Generalized Algebraic Data Types in Ocaml." https://cs242.stanford.edu/f17/assets/projects/2017/junjiek.pdf.

Laneve, Cosimo, Claudio Sacerdoti Coen, and Adele Veschetti. 2019. "On the Prediction of Smart Contracts' Behaviours." In *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, 397–415. https://doi.org/10.1007/978-3-030-30985-5/_23.

Maurizio Gabbrielli, Simone Martini. 2010. *Linguaggi Di Programmazione: Principi E Paradigmi*. 2nd ed. McGraw-Hill.

Poulsen, Casper Bach, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. "Intrinsically-Typed Definitional Interpreters for Imperative Languages." *PACMPL* 2 (POPL): 16:1–16:34. https://doi.org/10.1145/3158104.

Xavier Leroy, Alain Frisch, Damien Doligez, and Jérôme Vouillon. 2019. "The Ocaml System Release 4.09." http://caml.inria.fr/pub/docs/manual-ocaml.